Implementação de Escalonador de Tarefas Aperiódicas em RTOS

Arthur Menezes, Emanoel Aurélio Vianna Fabiano, Gabriell Alves de Araujo

Faculdade de Informática – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Av. Ipiranga, 6681 - Partenon, RS, 90619-900 – Porto Alegre – RS – Brazil

{arthur.aguiar, emanoel.fabiano, gabriell.araujo}@acad.pucrs.br

1. Introdução

Dentro do escopo da disciplina de Sistemas Embarcados, o presente artigo apresenta as modificações que foram realizadas no *Kernel*, assim como o algoritmo implementado, a organização para demonstrar o funcionamento da implementação e ao final uma análise sobre o desempenho do escalonador, incluindo parâmetros como j*itter* e *delay* para diferentes cenários.

2. Tarefas desenvolvidas

Em um modo geral, as tarefas desenvolvidas ao longo do trabalho, possuem como objetivo principal, ao final, implementar o suporte ao escalonamento de tarefas aperiódicas. Para isso, as seguintes modificações foram realizadas sobre o *Kernel*:

1. *kernel.h* - adicionar um ponteiro para a fila ou lista de tarefas aperiódicas;

```
struct queue *krnl_ap_queue; /*!< pointer to a queue of aperiodic tasks */

typedef struct {
       int id;
       uint32_t arrival_time;
       uint32_t release_time;

       uint32_t delay_time;

}task_time;

task_time task_time_array[MAX_TASKS];
int task_time_array_count;
#ifndef _delay_
#define _delay_
```

```
typedef struct node
{
        uint32 t delay value;
        struct node* next;
} delay node;
delay_node* delay_head;
delay_node* delay_tail;
int static size_delay_queue_MASTER = 0;
#endif
```

2. *main.c* - inicializar a fila / lista junto com as outras já existentes;

```
krnl_ap_queue = hf_queue_create(MAX_TASKS);

if (krnl_ap_queue == NULL) panic(PANIC_OOM);
```

3. *task.c* - modificação *hf_spawn()* e *hf_kill()* para funcionar com tarefas aperiódicas;

```
}else if (period == 0 && capacity > 0 && deadline == 0){
    if (hf_queue_addtail(krnl_ap_queue,krnl_task)) panic(PANIC_CANT_PLACE_AP);
```

...

```
} else if (krnl_task->period == 0 && krnl_task->capacity > 0 &&
krnl_task->deadline == 0) {
    k=hf_queue_count(krnl_ap_queue);
    for (i = 0; i < k; i++)
          if (hf_queue_get(krnl_ap_queue, i) == krnl_task) break;
     if (!k || i == k)  panic(PANIC_NO_TASKS_AP);
     for (j = i; j > 0; j--)
          if (hf_queue_swap(krnl_ap_queue, j, j-1))  panic(PANIC_CANT_SWAP);
     krnl_task2 = hf_queue_remhead(krnl_ap_queue);
} else{
```

...

```
if(add_task_time(i) == 1)  {
      task_time aux = get_task_time(i);
      if(aux.id != -1)  {
       aux.arrival_time = _readcounter();
      }
```

```
                kprintf("\n\n\n   TAREFA.ARRIVAL_TIME   =>   %d   ----\n\n\n",
aux.arrival_time);
}
```

...

```
int add_task_time(int id)  {
     if(task_time_array_count < MAX_TASKS) {
       task_time aux;
       aux.id = id;
       task_time_array[task_time_array_count] = aux;
       task_time_array_count++;
       return 1;
     } else {
       return 0;
     }
}
int get_task_time(int id_aux)  {
    int i;
    for(i=0; i<task_time_array_count; i++)  {
       if(task_time_array[i].id == id_aux) {
            return i;
       }
     }
     aux.id = -1;
     return -1;
}
```

4. *scheduler.c* - modificação para incluir o servidor aperiódico;

```
void add_to_delay_queue(delay_node** node_aux)
{
       //adiciona o nodo na primeira posição se a fila está vazia
       if(size_delay_queue_MASTER == 0)
       {
              delay_head = *node_aux;
              delay tail = delay head;
              size_delay_queue_MASTER++;
              return;
       }
```

```
        //adiciona o nodo na última posição se a fila não está vazia
        else
        {
                delay_tail->next = *node_aux;
                delay_tail = delay_tail->next;
                size_delay_queue_MASTER++;
                return;
        }
}
void print_jitter()
{
   delay_node* node_aux = delay_head;
   uint32_t sum = 0;
   uint32_t min = (*node_aux).delay_value;
   uint32_t max = (*node_aux).delay_value;
   uint32_t aux = 0;
   node_aux = (*node_aux).next;

   int i;
   for(i=1; i<size_delay_queue_MASTER; i++)
   {
      aux = (*node_aux).delay_value;
      node_aux = (*node_aux).next;
      sum += aux;

      if(aux < min)
      {
         min = aux;
      }
      if(aux > max)
      {
         max = aux;
      }
   }
   kprintf("QUANTIDADE DE DELAYS=%d\n", size_delay_queue_MASTER);
   kprintf("DELAY MINIMO=%d\n", min);
   kprintf("DELAY MAXIMO=%d\n", max);
   kprintf("JITTER=%d\n", max-min);
}

int32_t sched_aperiodic(void)
{
```

```
        //printf("ENTROU NO SCHEDULER sched_aperiodic!!!\n");
        int32_t k;
        uint16_t id = 0;
        int aux_id;

        while(hf_queue_count(krnl_ap_queue) > 0) {
                ap_queue_next();
                        krnl_task->capacity_rem--;
                        krnl_task->apjobs++;
                        aux_id = get_task_time(krnl_task->id);
                        task_time_array[aux_id].release_time = _readcounter();
                        task_time_array[aux_id].delay_time =
task_time_array[aux_id].release_time - task_time_array[aux_id].arrival_time;
                        delay_node* node_aux =  (delay_node*)
malloc(sizeof(delay_node));
                        node_aux->delay_value = task_time_array[aux_id].delay_time;
                        add_to_delay_queue(&node_aux);
                        return krnl_task->id;
                } else {
                        hf_kill(krnl_task->id);
                }
        }
        return 0;
}
```

3. Demonstração do funcionamento

Para garantir o correto funcionamento sobre a implementação do novo servidor, foi descrito uma aplicação de teste como segue abaixo:

1. Uma tarefa que realiza o disparo (criação) de tarefas aperiódicas entre 50 ms e 500 ms aproximadamente;
2. Aplicação de tarefas de tempo real periódicas relacionadas com outras tarefas.

```
#include <hellfire.h>
#define TOTAL SIMULATION TIME 5000
int RUN_REPORT=1;
int static count id=0;
int IS_RUNNING=1;
```

```
void caso_de_teste_1(void);
void caso_de_teste_2(void);
void simple_task(void);
void lancar_tarefas_aperiodicas(void);
void caso_de_teste_1(void)
{
   hf_spawn(lancar_tarefas_aperiodicas, 0, 0, 0, "aperiodic tasks creator", 1024);
}

void caso de teste 2(void)
{
   hf spawn(lancar tarefas aperiodicas, 0, 0, 0, "aperiodic tasks creator", 1024);
   hf_spawn(simple_task, 60, 20, 60, "task a", 1024);
   hf_spawn(simple_task, 0, 90, 0, "task b", 1024);
   hf_spawn(simple_task, 0, 0, 0, "task c", 1024);
   hf_spawn(simple_task, 70, 30, 70, "task d", 1024);
   hf_spawn(simple_task, 0, 50, 0, "task e", 1024);
}

void lancar_tarefas_aperiodicas(void){
   int time to wait;
   while(IS_RUNNING==1)
   {
       time_to_wait = (random()%450)+50;
       delay ms(time to wait);
       hf_spawn(simple_task, 0, (random()%100)+10, 0, "aperiodic simple_task", 1024);
   }
}

void simple_task(void){
       int i, x = 0;
       while(IS_RUNNING==1) {
           for(i=0; i<1000; i++) {
               x++;
           }
           for(i=1000; i>0; i--)
           {
               x--;
           }
       }
}
```

```
void simulation control(void){
   if(RUN_REPORT == 1)
   {
       delay_ms(TOTAL_SIMULATION_TIME);
       kprintf("\n\n\n\n\n");
       kprintf("RELATORIO DA SIMULACAO ---- INICIO\n");
       print jitter();
       kprintf("RELATORIO DA SIMULACAO ---- FIM\n");
       RUN REPORT=0;
       IS_RUNNING=0;
   }
}

void app_main(void){
   //IMPRIME RELATORIO APOS ESGOTAR O TEMPO DE SIMULACAO
   hf_spawn(simulation_control, 0, 0, 0, "simulation control", 1024);
   //caso_de_teste_1();
   caso_de_teste_2();
   return;
}
```

A validação da especificação em relação ao desempenho se dá através da utilização de um temporizador implementado junto ao *hardware*.

4. Análise do Desempenho do Escalonador

Após um estudo e implementação do escalonador, pode-se concluir que tais técnicas possuem um papel de grande importância em diversos espaços da área de Sistemas Embarcados, com o objetivo de garantir que sistemas de hardware possam escolher a melhor alternativa entre as existentes para cumprir a tarefa atual.

Para a realização das análises, alguns procedimentos prévios foram efetuados, tais como a elaboração de conjuntos de tarefas a serem processadas, variando parâmetros de acordo com as intenções da equipe para com a análise e a definição de uma estrutura de dados própria para a realização do armazenamento dos dados temporais acerca de cada tarefa escalonada. Ao final, os dados representam informações como:

- *Arrival time*: *timestamp* (momento no tempo) no qual a tarefa foi recebida;
- *Release time*: *timestamp* (momento no tempo) no qual a tarefa foi escalonada;
- *Delay time*: quantidade de tempo entre o *arrival time* e o *release time* de cada

tarefa.

Com isso, foram definidos os conjuntos de tarefas apresentados nas subseções a seguir.

4.1. Primeiro Caso de Teste

Para o primeiro caso de teste, o seguinte conjunto foi definido:

Conjunto de Tarefas 1			
Nome	**Period**	**Capacity**	**Deadline**
aperiodic simple_task	0	*random value between* 10 *and* 110	0

Após o processamento, os seguintes resultados foram obtidos:

Medida	**Valor (em ciclos)**
Quantidade de escalonamentos	383
Delay mínimo	245585
Delay máximo	21480693
Jitter	21480693 - 245585 = 21235108

4.2. Segundo Caso de Teste

Para o segundo caso de teste, um conjunto com mais de uma tarefa foi foi definido:

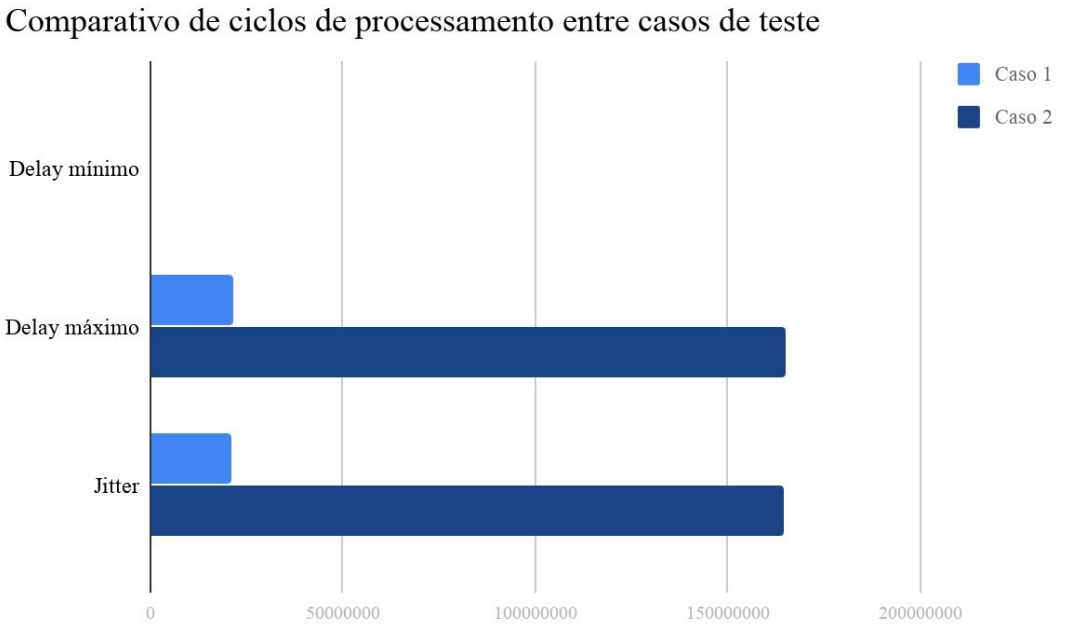
Conjunto de Tarefas 2			
Nome	**Period**	**Capacity**	**Deadline**
task a	60	20	60
task b	0	90	0
task c	0	0	0
task d	70	30	70
task e	0	50	0
task f	0	0	0

Após o processamento, os seguintes resultados referentes foram obtidos:

Medida	**Valor (em ciclos)**
Quantidade de escalonamentos	227
Delay mínimo	245190
Delay máximo	165001440
Jitter	165001440 - 245190 = 164756250

4.3. Comparando os Casos de Teste

A partir da obtenção dos resultados para ambos os casos de teste, pode-se realizar um comparativo entre eles e, assim, analisar a diferença que houve no processamento. Para tal, pode-se observar o gráfico de barras a seguir.



Como pode-se observar, o segundo caso de teste, que envolve o processamento de duas tarefas aperiódicas, intercalando-as com tarefas de diferentes tipos, é 6,76 vezes mais demorado que o primeiro caso de teste. Além de tudo isso, deve-se levar em conta também que, para o segundo caso de teste, houve uma quantidade inferior de tarefas escalonadas.

5. Conclusão

Após um estudo e implementação do escalonador, pode-se concluir que tais técnicas possuem um papel de grande importância em diversos espaços da área de Sistemas Embarcados, com o objetivo de garantir que sistemas de hardware possam escolher a melhor alternativa entre as existentes para cumprir a tarefa atual. O projeto foi desenvolvido em equipe, utilizando-se versionamento (neste caso, o Git), a partir de um *fork* do repositório do professor. O repositório utilizado para o projeto pode ser encontrado através do seguinte endereço de web: https://github.com/arthuramsouza/hellfireos/.

Após a implementação do escalonador aperiódico e sua alocação, no que se refere ao *kernel* e ao escopo do sistema operacional de tempo-real em si, entre os escalonadores

real-time e *best effort*, foi realizado um processo de definição dos casos de teste a serem executados. Com a definição dos conjuntos de tarefas de cada casos de teste, cada caso foi definido no arquivo trabalho_1.c presente no sub-diretório /app/trabalho_1/. Após os processamentos das tarefas, foi possível observar que, dependentemente aos parâmetros aplicados às tarefas e a quantidade, os casos de teste apresentaram uma relevante diferença em seus resultados, conforme demonstrado na subseção 4.3.